[Audience looks around]
“What is going on? There must be some context we’re missing”
Subsets of strings are called *languages*

Examples:
- $\Sigma^* =$ All strings over alphabet $\Sigma$
- Palindromes over $\Sigma$
- Binary strings that don’t have a 0 after a 1
- Binary strings with an equal # of 0’s and 1’s
- Legal variable names in Java/C/C++
- Syntactically correct Java/C/C++ programs
- Valid English sentences
Regular expressions over $\Sigma$

- **Basis:**
  
  $\varepsilon$ is a regular expression (could also include $\emptyset$)
  
  $a$ is a regular expression for any $a \in \Sigma$

- **Recursive step:**
  
  If $A$ and $B$ are regular expressions then so are:
  
  $A \cup B$
  
  $AB$
  
  $A^*$
Last class: Regular Expression is a “pattern”

\( \epsilon \) matches the **empty string**

\( a \) matches the one character string \( a \)

\( A \cup B \) matches all strings that either \( A \) matches or \( B \) matches (or both)

\( AB \) matches all strings that have a first part that \( A \) matches followed by a second part that \( B \) matches

\( A^* \) matches all strings that have any number of strings (even 0) that \( A \) matches, one after another

Yields a **language** = the set of strings matched by the regular expression
<table>
<thead>
<tr>
<th>Regular Expression</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>001*</td>
<td>{00, 001, 0011, 00111, ...}</td>
</tr>
<tr>
<td>0<em>1</em></td>
<td>{Binary strings with any number of 0s followed by any number of 1s}</td>
</tr>
<tr>
<td>(0 ∪ 1) 0 (0 ∪ 1) 0</td>
<td>{0000, 1000, 0010, 1010}</td>
</tr>
<tr>
<td>(0<em>1</em>)*</td>
<td>{All binary strings}={0,1}*</td>
</tr>
<tr>
<td>(0 ∪ 1)*</td>
<td>{All binary strings}={0,1}*</td>
</tr>
<tr>
<td>(0 ∪ 1)* 0110 (0 ∪ 1)*</td>
<td>{All binary strings containing substring 0110}</td>
</tr>
</tbody>
</table>
Regular Expressions in Practice

- Used to define the *tokens* of a programming language
  - legal variable names, keywords, etc.

- Used in `grep`, a program that does pattern matching searches in UNIX/LINUX

- We can use regular expressions in programs to process strings!
Regular Expressions in Java

Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaaab");
boolean b = m.matches();

[01] a 0 or a 1 ^ start of string $ end of string
[0-9] any single digit \ . period \ , comma \ − minus
. any single character
ab a followed by b (AB)
(a|b) a or b (A ∪ B)
a? zero or one of a (A ∪ ε)
a* zero or more of a A*
a+ one or more of a AA*

• e.g. ^[\−+]?[0−9]∗(\ . | \ , )?[0−9]+$ General form of decimal number e.g. 9.12 or -9,8 (Europe)
Examples

• All binary strings that have an even # of 1’s
Examples

• All binary strings that have an even # of 1’s

e.g., $0^* (10^*10^*)^*$
Examples

- All binary strings that have an even # of 1’s
  
  e.g., $0^* (10^*10^*)^*$

- All binary strings that don’t contain 101
Examples

- All binary strings that have an even # of 1’s
  
  e.g., \( 0^* (10^*10^*)^* \)

- All binary strings that *don’t* contain 101
  
  e.g., \( 0^* (1 \cup 000^*)^* 0^* \)

  at least two 0s between 1s
Limitations of Regular Expressions

- Not all languages can be specified by regular expressions
- Even some easy things like
  - Palindromes
  - Strings with equal number of 0’s and 1’s
- But also more complicated structures in programming languages
  - Matched parentheses
  - Properly formed arithmetic expressions
  - etc.
Context-Free Grammars

• A Context-Free Grammar (CFG) is given by a finite set of substitution rules involving
  – Alphabet $\Sigma$ of *terminal symbols* that can’t be replaced
  – A finite set $V$ of *variables* that can be replaced
  – One variable, usually $S$, is called the *start symbol*

• The substitution rules involving a variable $A$, written as

$$A \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k$$

where each $w_i$ is a string of variables and terminals
  – that is $w_i \in (V \cup \Sigma)^*$
How CFGs generate strings

• Begin with “S”

• If there is some variable $A$ in the current string, you can replace it by one of the $w$’s in the rules for $A$
  - $A \rightarrow w_1 \mid w_2 \mid \cdots \mid w_k$
  - Write this as $xAy \Rightarrow xwy$
  - Repeat until no variables left

• The set of strings the CFG describes are all strings, containing no variables, that can be generated in this manner after a finite number of steps
Example Context-Free Grammars

Example: \( S \rightarrow 0S \mid S1 \mid \varepsilon \)
Example Context-Free Grammars

Example: \[ S \rightarrow 0S \mid S1 \mid \varepsilon \]

\[ 0^*1^* \]
Example Context-Free Grammars

Example: \( S \rightarrow 0S \mid S1 \mid \varepsilon \)

\[0^*1^*\]

Example: \( S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon \)
Example Context-Free Grammars

Example: $S \rightarrow 0S \mid S1 \mid \varepsilon$

$0^*1^*$

Example: $S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon$

The set of all binary palindromes
Example Context-Free Grammars

Grammar for \( \{0^n1^n : n \geq 0\} \)
(i.e., matching \(0^*1^*\) but with same number of 0’s and 1’s)
Example Context-Free Grammars

Grammar for \( \{0^n1^n : n \geq 0\} \)
(i.e., matching \(0^*1^*\) but with same number of 0’s and 1’s)

\[
S \rightarrow 0S1 | \varepsilon
\]
Example Context-Free Grammars

Grammar for \( \{0^n1^n : n \geq 0\} \)
(i.e., matching \( 0^*1^* \) but with same number of 0’s and 1’s)

\[
S \rightarrow OS1 \mid \varepsilon
\]

Grammar for \( \{0^n1^{2n} : n \geq 0\} \)
Example Context-Free Grammars

Grammar for $\{0^n 1^n : n \geq 0\}$
(i.e., matching $0^* 1^*$ but with same number of 0’s and 1’s)

$$S \rightarrow 0S1 \mid \varepsilon$$

Grammar for $\{0^n 1^{2n} : n \geq 0\}$

$$S \rightarrow 0S11 \mid \varepsilon$$
Example Context-Free Grammars

Grammar for \( \{0^n1^n: n \geq 0\} \)
(i.e., matching \(0^*1^*\) but with same number of 0’s and 1’s)

\[ S \rightarrow 0S1 \mid \varepsilon \]

Grammar for \( \{0^n1^{n+1}0: n \geq 0\} \)
Example Context-Free Grammars

Grammar for \( \{0^n 1^n : n \geq 0 \} \)
(i.e., matching \( 0^* 1^* \) but with same number of 0’s and 1’s)

\[
S \rightarrow 0S1 \mid \varepsilon
\]

Grammar for \( \{0^n 1^{n+1} 0 : n \geq 0 \} \)

\[
S \rightarrow A 10
A \rightarrow 0A1 \mid \varepsilon
\]
Example Context-Free Grammars

Example: $S \rightarrow (S) \mid SS \mid \varepsilon$
Example Context-Free Grammars

Example: \[ S \rightarrow (S) \mid SS \mid \varepsilon \]

The set of all strings of matched parentheses
Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s (not just $0^n1^n$, also 0101, 0110, etc.)
Example Context-Free Grammars

Binary strings with equal numbers of 0s and 1s
(not just \(0^n1^n\), also 0101, 0110, etc.)

\[
S \rightarrow SS \mid 0S1 \mid 1SO \mid \varepsilon
\]

An easy structural induction can show that everything generated by \(S\) has an equal \# of 0s and 1s

Why does this generate all such strings?
Example Context-Free Grammars

Let $x \in \{0,1\}^*$. Define $f_x(k)$ to be the number of 0s minus the number of 1s in the first $k$ characters of $x$.

E.g., for $x = 011100$

$$f_x(k) = 0 \text{ when first } k \text{ characters have } \#0s = \#1s$$

- starts out at 0 $f_x(0) = 0$
- ends at 0 $f_x(n) = 0$
Three possibilities for $f_x(k)$ for $k \in \{1, \ldots, n - 1\}$

- $f_x(k) > 0$ for all such $k$
  
  $$S \rightarrow 0S1$$

- $f_x(k) < 0$ for all such $k$
  
  $$S \rightarrow 1S0$$

- $f_x(k) = 0$ for some such $k$
  
  $$S \rightarrow SS$$
Simple Arithmetic Expressions

\[
E \rightarrow E + E \mid E \times E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \\
\mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]

Generate \((2 \times x) + y\)
Simple Arithmetic Expressions

\[ E \rightarrow E+E \mid E*E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

Generate \((2*x) + y\)

\[ E \Rightarrow E+E \Rightarrow (E)+E \Rightarrow (E*E)+E \Rightarrow (2*E)+E \Rightarrow (2*x)+E \Rightarrow (2*x)+y \]
Suppose that grammar $G$ generates a string $x$

- A *parse tree* of $x$ for $G$ has
  - Root labeled $S$ (start symbol of $G$)
  - The children of any node labeled $A$ are labeled by symbols of $w$ left-to-right for some rule $A \rightarrow w$
  - The symbols of $x$ label the leaves ordered left-to-right

```
S → 0S0 | 1S1 | 0 | 1 | ε
```

Parse tree of 01110
Simple Arithmetic Expressions

\[ E \rightarrow E + E \mid E \cdot E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

Generate \( x + y \cdot z \) in two ways that give two different parse trees
Simple Arithmetic Expressions

\[ E \rightarrow E + E \mid E * E \mid (E) \mid x \mid y \mid z \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \]

Generate \( x+y*z \) in ways that give two different parse trees

- \( E \Rightarrow E + E \Rightarrow x + E \Rightarrow x + E * E \Rightarrow x + y * E \Rightarrow x + y * z \) (add \( x \) to the product of \( y \) and \( z \))
- \( E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow x + E * E \Rightarrow x + y * E \Rightarrow x + y * z \) (add \( x \) to \( y \), then multiply by \( z \))
building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term
- **F** – factor
- **I** – identifier
- **N** – number

E → T | E+T
T → F | F*T
F → (E) | I | N
I → x | y | z
N → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

No longer allows:

```
E
  /\  *
 /   \\
E    E
  /\  +
 /   \\
E    E
    /\  \
   /   \\
  x    y
```
building precedence in simple arithmetic expressions

- $E$ – expression (start symbol)
- $T$ – term
- $F$ – factor
- $I$ – identifier
- $N$ – number

$E \rightarrow T \mid E + T$

$T \rightarrow F \mid F \ast T$

$F \rightarrow (E) \mid I \mid N$

$I \rightarrow x \mid y \mid z$

$N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$
building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term  **F** – factor  **I** – identifier  **N** – number

  \[
  E \rightarrow T \mid E + T \\
  T \rightarrow F \mid F * T \\
  F \rightarrow (E) \mid I \mid N \\
  I \rightarrow x \mid y \mid z \\
  N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
  \]

Still allows:

```
E + E
  \[
  E \rightarrow x \mid E \ast E \\
  E \ast E \rightarrow y \mid z
  \]
```
building precedence in simple arithmetic expressions

- **E** – expression (start symbol)
- **T** – term  **F** – factor  **I** – identifier  **N** - number

\[
E \rightarrow T \mid E + T \\
T \rightarrow F \mid F * T \\
F \rightarrow (E) \mid I \mid N \\
I \rightarrow x \mid y \mid z \\
N \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\]
CFGs and recursively-defined sets of strings

• A CFG with the start symbol $S$ as its *only* variable recursively defines the set of strings of terminals that $S$ can generate

• A CFG with more than one variable is a simultaneous recursive definition of the sets of strings generated by *each* of its variables
  – sometimes necessary to use more than one
Theorem: For any set of strings (language) $A$ described by a regular expression, there is a CFG that recognizes $A$.

Proof idea:
$P(A)$ is “$A$ is recognized by some CFG”
Structural induction based on the recursive definition of regular expressions...
Regular Expressions over $\Sigma$

• **Basis:**
  
  – $\varepsilon$ is a regular expression
  – $a$ is a regular expression for any $a \in \Sigma$

• **Recursive step:**
  
  – If $A$ and $B$ are regular expressions then so are:
    
    $A \cup B$
    
    $AB$
    
    $A^*$
CFGs are more general than REs

- CFG to match RE $\varepsilon$
  
  $$S \rightarrow \varepsilon$$

- CFG to match RE $a$ (for any $a \in \Sigma$)
  
  $$S \rightarrow a$$
CFGs are more general than REs

Suppose CFG with start symbol $S_A$ matches RE $A$
CFG with start symbol $S_B$ matches RE $B$

- CFG to match RE $A \cup B$
  
  \[ S \rightarrow S_A \mid S_B \]  
  + rules from original CFGs

- CFG to match RE $AB$
  
  \[ S \rightarrow S_A S_B \]  
  + rules from original CFGs
CFGs are more general than REs

Suppose CFG with start symbol $S_A$ matches RE $A$

- CFG to match RE $A^*$ ($= \varepsilon \cup A \cup AA \cup AAA \cup ...$)

$$S \rightarrow S_A S \mid \varepsilon$$

+ rules from CFG with $S_A$
Backus-Naur Form (The same thing...)

BNF (Backus-Naur Form) grammars

– Originally used to define programming languages
– Variables denoted by long names in angle brackets, e.g.
  <identifier>, <if-then-else-statement>, <assignment-statement>, <condition>

  ::=  used instead of  →
BNF for C

statement:  
  ((identifier | "case" constant-expression | "default") ":")* 
  (expression? ";" | 
   block | 
   "if" "(" expression ")" statement | 
   "if" "(" expression ")" statement "else" statement | 
   "switch" "(" expression ")" statement | 
   "while" "(" expression ")" statement | 
   "do" statement "while" "(" expression ")" ";" | 
   "for" "(" expression? ";" expression? ";" expression? ")" statement | 
   "goto" identifier ";" | 
   "continue" ";" | 
   "break" ";" | 
   "return" expression? ";" 
  )

block: "{" declaration* statement* "}"

expression:
  assignment-expression%

assignment-expression: ( 
  unary-expression ( 
    ":=" | ":*=" | ":/=" | ":%=" | ":+=" | ":-=" | ":<=" | ":>=" | ":&=" | 
    ":^=" | ":|=" 
  )
) * conditional-expression

conditional-expression:
  logical-OR-expression ( "?:" expression ":" conditional-expression )?
Back to middle school:

\[ <\text{sentence}> ::= <\text{noun phrase}> <\text{verb phrase}> \]
\[ <\text{noun phrase}> ::= <\text{article}> <\text{adjective}> <\text{noun}> \]
\[ <\text{verb phrase}> ::= <\text{verb}> <\text{adverb}> | <\text{verb}> <\text{object}> \]
\[ <\text{object}> ::= <\text{noun phrase}> \]

Parse:

The yellow duck squeaked loudly
The red truck hit a parked car